

# PLSQL

*Maggie LEKPA*

# Introduction

## SQL : Structured Query Language

Langage structuré de requête

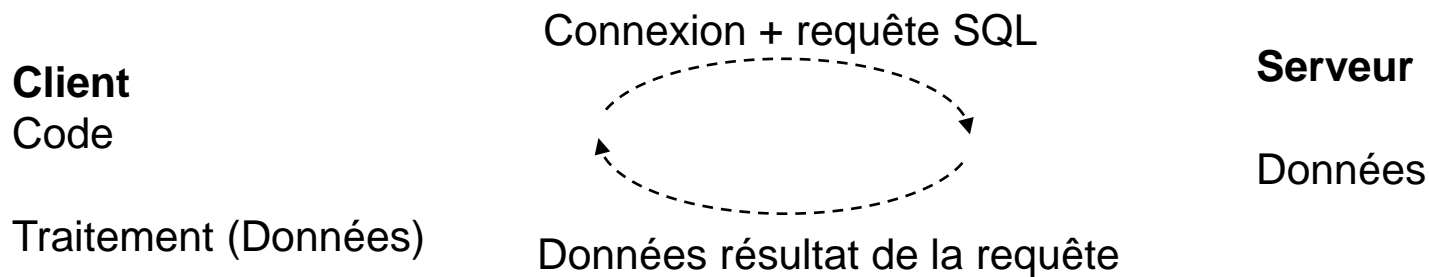
Le langage SQL est un langage déclaratif et non un langage de « programmation » qui permet d'effectuer des requêtes dans un langage **simple**.

Ne permet pas de définir des fonctions ou des variables ou d'effectuer une itération...

# Introduction

**SQL** : Structured Query Language  
Langage structuré de requête

**Comment ça se passe ?**



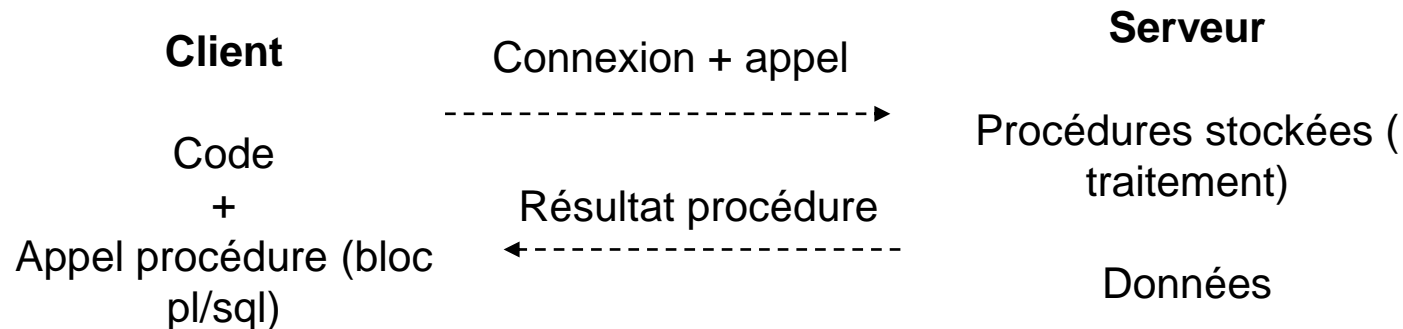
Le traitement des données se passe au niveau de l'application cliente

Multiples connexion au serveur

# Introduction

## PL/SQL : Programming Langage / Structured Query Langage

PL/SQL est un langage procédural qui intègre SQL. Il permet de définir un ensemble de commandes contenues dans des blocs pl/sql;



Traitement des données se fait côté serveur. Volume de données réduit entre le client et le serveur. Code PL/SQL portable

# Introduction

**PL/SQL** : Programming Langage / Structured Query Langage

## **Coté serveur :**

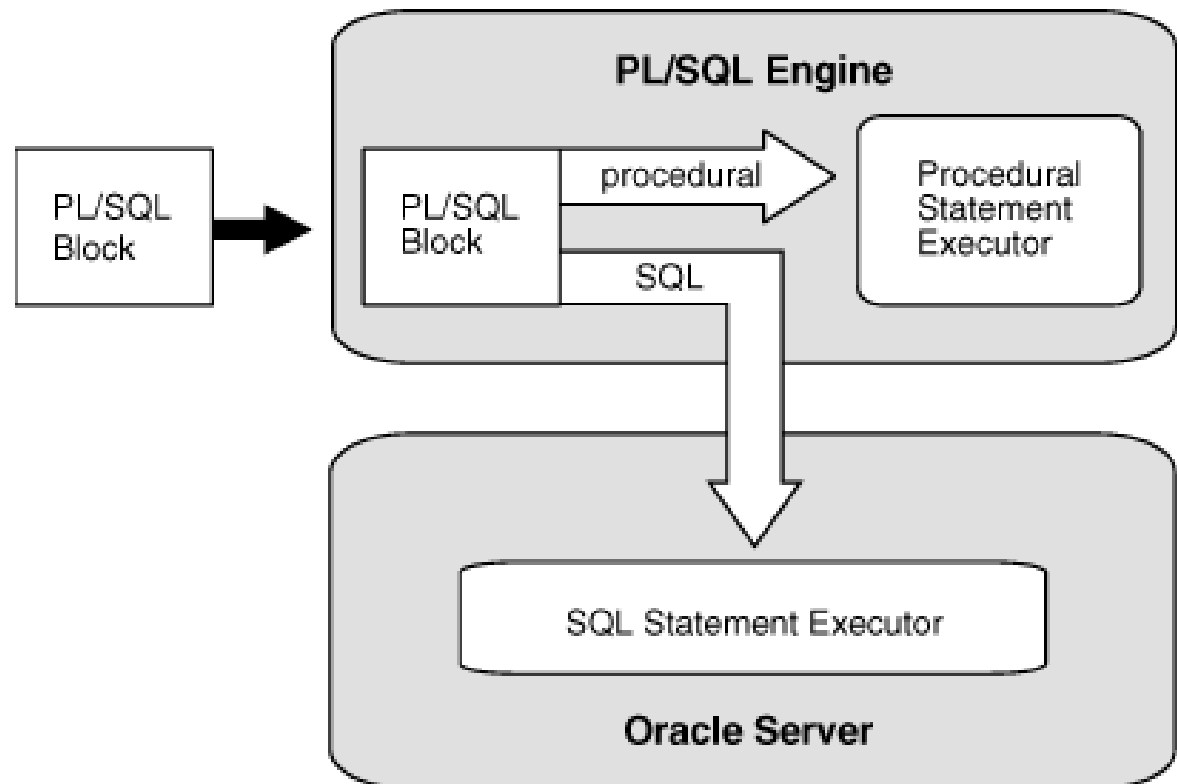
- Blocs d'instructions anonymes et non persistants
- Procédures et fonctions : persistants
- Triggers : déclencheurs sur DML-DDL-ERREUR
- Paquetages :persistants

## **Coté client :**

Outils clients utilisant PL/SQL

# Introduction

- Séparation des ordres SQL et PL/SQL
- Passage des requêtes SQL au processeur SQL
- Passage des instructions procédurales au processeur d'instructions procédurales



# BLOCS PL/SQL

# Les blocs PL/SQL

Un bloc PL/SQL est l'unité de programmation PL/SQL

Deux types de blocs :

- **Bloc anonyme** : ensemble d'instructions qui s'exécute à l'endroit où il existe.
- **Bloc nommé** : procédure ou une fonction, pouvant être appelées autant de fois que nécessaire.

Parties d'un bloc PL/SQL :

- **Partie declarative** (facultative): permet de déclarer les variables et de les initialiser; ne contient pas d'exécutable...
- **Partie d'exécution** (obligatoire) : contient les instructions d'exécution
- **Partie de gestion des erreurs** (facultative) : contient le code à exécuter en cas d'erreur.



# Les blocs PL/SQL

## Structure d'un bloc

*[<Entête de bloc>]* (valable pour les fonctions, procédures package)

*[DECLARE*

*Constantes*

*,variables*

*,Cursors]*

*BEGIN*

*Instructions*

*EXCEPTION*

*gestion des erreurs*

*END;*

# Les blocs PL/SQL

Les blocs PL/SQL peuvent être imbriqués

*[<Entête de bloc>]*

*[DECLARE*

*BEGIN*

*DECLARE*

*BEGIN*

*DECLARE*

*BEGIN*

*...*

*END*

*END;*

*END;*

# Les blocs PL/SQL

## Exemple

```
DECLARE v_variable VARCHAR(255):= 'bienvenu au premier cours de BD';
```

```
BEGIN
```

```
DBMS_OUTPUT.PUT_LINE(v_variable);
```

```
END;
```

---

```
bienvenu au premier cours de BD
```

```
Statement processed.
```

# TYPES DE DONNEES

# Type de données

## ➤ Types scalaires

- CHAR(taille) : chaîne de caractère de longueur fixe, 2000 max
- VARCHAR2(taille) : chaîne de caractère de longueur variable (4000 max)
- NCHAR et NVARCHAR2 : pour les caractères unicode
- NUMBER: numérique positif et négatif. A pour sous type INT, SMALLINT, REAL, DECIMAL
- DATE
- BOOLEAN: TRUE, FALSE, NULL

# Type de données

## ➤ Type implicite

Le type implicite fait reference à une entité déjà existante.

**%TYPE** permet de faire reference à un type existant

**%ROWTYPE** permet de faire reference à la structure d'une table existante

### Exemple :

Variable de même type que la colonne Name de la table Etudiant

**DECLARE V\_variable Etudiant.Name%type;**

Variable faisant reference à une structure entiere d'une table

**DECLARE V\_Rec Employe%ROWTYPE**

# Type de données

## ➤ Types définis par l'utilisateur

Exemple :

```
DECLARE SUBTYPE Type_Date IS DATE;  
V_var Type_Date;  
BEGIN  
SELECT sysdate INTO V_var FROM DUAL;  
DBMS_OUTPUT.PUT_LINE(V_var);  
END;
```

```
SQL> DECLARE SUBTYPE Type_Date IS DATE;  
2 V_var Type_Date;  
3 BEGIN  
4 SELECT sysdate INTO V_var FROM DUAL;  
5 DBMS_OUTPUT.PUT_LINE(V_var);  
6 END;  
7 /  
27-SEP-22  
  
PL/SQL procedure successfully completed.
```

# Type de données composés

## Types composés : Record et Table

➤ RECORD : permet de définir un enregistrement

### 1. Déclare le type

*DECLARE TYPE First\_Record IS RECORD*

*(Nom VARCHAR2(50)*

*,Age INT);*

### 2. Déclare une variable de type First\_Record

*DECLARE V\_V First\_Record;*

### 3. Affectation des valeurs à la variable

*V\_V.Nom := 'PAM';*

*V\_V.age:=20;*



# Type de données

## ➤ Table

C'est un vecteur d'éléments de même type (**non composé**) accessible au moyen d'un indice préalablement déclaré.

### Syntaxe

*DECLARE TYPE Nom\_Type IS TABLE OF type\_scalaire [NOT NULL] INDEX BY  
[BINARY\_INTEGER / PLS\_INTEGER / VARCHAR2(size limit)]*

### Exemple

```
DECLARE TYPE Type_Table IS TABLE OF VARCHAR2(50) INDEX BY BINARY_INTEGER;  
First_Table Type_Table ;  
BEGIN  
First_Table(1):= 'Première Valeur'  
First_Table(2):= 'Seconde Valeur'  
END;
```

# Type de données

Manipulation du type compose TABLE

**TableName.count** : nombre d'élèments

**TableName.Exists(i)** = TRUE si i-ième élément existe

**TableName.First** et **TableName.Last** : indice du premier et dernier élément si pas vide

**TableName.next(i)** et **TableName.prior(i)**: renvoie l'indice qui suit ou precede la ième case. Vaut null si elle est vide

**TableName.delete** : supprime tous les éléments

**TableName.extend(p)** : allonge la table de p éléments.

# Type de données

## Exemple

```
DECLARE TYPE First_Table IS TABLE OF VARCHAR2(50) INDEX BY VARCHAR2(50);  
V_First_Table First_Table;  
BEGIN  
V_First_Table(1):='Première Valeur';  
V_First_Table(2):='Seconde Valeur';  
V_First_Table(4):='Quatrieme Valeur';  
DBMS_OUTPUT.PUT_LINE(V_First_Table.count);  
DBMS_OUTPUT.PUT_LINE(V_First_Table(4));  
DBMS_OUTPUT.PUT_LINE(V_First_Table.next(1));  
END
```

```
3  
Seconde Valeur  
2
```

```
Statement processed.
```

# Déclaration des constantes

## Constantes locales

### Syntaxe :

DECLARE Nom\_Variable [CONSTANT] type [NOT NULL]:=expression

Le mot clé **CONSTANT** permet de définir une constante; une initialization est obligatoire et la valeur ne pourra être change.

### Exemple :

```
DECLARE V_variable1 VARCHAR2(5);
```

```
DECLARE V_variable2 VARCHAR2(5):='OK'
```

```
DECLARE V_variable2 CONSTANT VARCHAR2(5):='OK'
```

# Déclaration des objets

Pensez a bien préfixé vos objects pour une bonne lisibilité :

Variables : **V**\_Nom\_Variable

Exception : **E**\_Nom\_Exception

Curseur : **C**\_Nom\_Curseur

Paramètre : **P**\_Nom\_Paramètre

# CONVERSION

## ➤ Conversion explicite

Utilisation des fonction prédéfinies (TO\_NUMBER, TO\_CHAR, TO\_DATE...)

```
DECLARE V_number NUMBER(10,5) :=TO_NUMBER('15,40');
```

## ➤ Conversion implicite

```
DECLARE V_number NUMBER(10,5) :='15,40' —conversion implicite de la chaine de caractère
```

**! Conseillé de faire des conversions explicites**

# Commentaires

- Commentaires multi-lignes

```
/* Je
```

```
Suis un commentaire
```

```
Multi-lignes*/
```

- Commentaire sur une ligne

```
-- Je suis un commentaire sur une ligne
```

# EXCEPTION



# EXCEPTIONS

Lors du traitement d'un bloc PL/SQL, une erreur peut se produire; ce sont des exceptions. Lorsqu'elles ne sont pas traitées, elles provoquent l'échec du bloc PL/SQL.

La gestion des exceptions se fait dans la section EXCEPTION du bloc PL/SQL et permet de transformer un échec en succès.

**Exceptions internes:** ce sont celles détectées implicitement par ORACLE. Elles sont de la forme ORA-XXX.

**Exceptions externes:** détectées explicitement par le développeur; elles sont définies dans la section DECLARE.

# EXCEPTIONS

## Exemple :

```
DECLARE V_Result NUMBER;  
V_Num NUMBER := 100;  
V_Deno NUMBER := 0;  
BEGIN  
V_Result := V_Num / V_Deno ;  
END;
```

Résultat :

```
ORA-01476: le diviseur est égal à zéro
```

```
DECLARE V_Result NUMBER;  
V_Num NUMBER := 100;  
V_Deno NUMBER := 0;  
BEGIN  
V_Result := V_Num / V_Deno ;  
EXCEPTION  
WHEN ZERO_DIVIDE THEN  
DBMS_OUTPUT.PUT_LINE('division par zéro');  
END;
```

Résultat:

---

```
division par zéro
```

```
Statement processed.
```

# EXCEPTIONS

## Exceptions les plus courantes

- **DUP\_VAL\_ON\_INDEX (ORA-00001)** : le tuple existe déjà; si une table n'admet pas de doublons, l'ajout d'un tuple déjà existant lève cet exception.
- **NO\_DATA\_FOUND** : pas de données
- **TOO\_MANY\_ROWS** : retour de plusieurs ligne par le SELECT
- **VALUE\_ERROR** : n'est pas de même type ou NULL
- **ZERO\_DIVIDE** : division par zéro
- **INVALID\_CURSOR** : curseur n'est pas autorisé
- **INVALID\_NUMBER** : échec d'une conversion d'une chaîne de caractère en un nombre
- **OTHERS** : exceptions non définies

# EXCEPTIONS

Il est aussi possible de définir un message d'erreur.

**Syntaxe :** *RAISE\_APPLICATION\_ERROR(error\_number, message)*

Error\_number : nombre négatif dont la valeur absolue est comprise entre 20000 et 20999

**Exemple :**

*DECLARE*

*v\_empno NUMBER := 9999;*

*v\_sal NUMBER;*

*BEGIN*

*select salary into v\_sal from employe where Emp\_id = v\_empno;*

*DBMS\_OUTPUT.PUT\_LINE(v\_sal);*

*EXCEPTION*

*WHEN ZERO\_DIVIDE THEN*

*DBMS\_OUTPUT.PUT\_LINE('Division par zero');*

*WHEN OTHERS THEN*

*raise\_application\_error(-20102, 'Je suis une erreur définie');*

*END;*

ORA-20102: Je suis une erreur définie

# PROCEDURES

# Procédures

C'est le code PL/SQL compilé et stocké dans le dictionnaire Oracle.

## Syntaxe :

```
CREATE [OR REPLACE] PROCEDURE Nom_Procedure(P_p1 type, P_p2  
type, ...) IS
```

```
BEGIN
```

```
-- Instruction
```

```
EXCEPTION
```

```
--Gestion des erreurs
```

```
END Nom_Procedure;
```

REPLACE : remplace une procédure existante (suppression puis recréation)

# Procédures

## Différents modes des paramètres

- **IN** (mode par défaut) → en entrée, lecture seule
- **OUT** → en sortie, écriture
- **IN OUT** → en entrée/sortie , lecture et écriture

## Appel d'une procédure

*EXECUTE Nom\_Procedure(p1, p2...);*

*CALL Nom\_Procedure(p1,p2...);*

*BEGIN*

*Nom\_Procedure(p1,p2...)*

*END;*

# Procédures

**Exemple 1** : Compter le nombre d'employés de l'entreprise

```
CREATE OR REPLACE PROCEDURE Compte_Emp  
IS  
V_Nbre_Employe INT;  
BEGIN  
SELECT COUNT(*) INTO V_Nbre_Employe FROM Employe;  
DBMS_OUTPUT.PUT_LINE(CONCAT('Nombre employé =', V_Nbre_Employe));  
END Compte_Emp;
```

```
SQL> execute Compte_Emp  
Nombre employe =4  
  
PL/SQL procedure successfully completed.
```



# Procédures

Exemple 2 : Compter le nombre d'employé d'un département

```
CREATE OR REPLACE PROCEDURE Nbre_Emp_Dept(P_Dept_Name VARCHAR2)
IS
V_Nbre_Emp INT;
BEGIN
SELECT COUNT(*) INTO V_Nbre_Emp FROM Employe E
INNER JOIN Department D ON E.Dept_Id = D.Dept_ID
WHERE D.Dept_Name=P_Dept_Name;
DBMS_OUTPUT.PUT_LINE('Nombre employé du département '|| P_Dept_Name || ' est : ' ||
V_Nbre_Emp);
END Nbre_Emp_Dept;
```

```
SQL> EXECUTE Nbre_Emp_Dept('RESEARCH');
Nombre employe du departement RESEARCH est : 2
PL/SQL procedure successfully completed.
```

# Procédures

## Exemple 3: Afficher le salaire d'un employé

```
CREATE OR REPLACE PROCEDURE Show_Salary(P_Emp_Id Employee.Emp_Id%TYPE ,  
P_Salary OUT NUMBER)  
IS  
V_Emp_Name Employee.Emp_Nom%TYPE;  
BEGIN  
SELECT Salary, Emp_Nom INTO P_Salary , V_Emp_Name  
FROM Employee WHERE Emp_Id=P_Emp_Id;  
DBMS_OUTPUT.PUT_LINE('Le salaire de ' || V_Emp_Name || ' est de ' || P_Salary );  
END Show_Salary;
```

```
SQL> DECLARE P_sal NUMBER;  
2 BEGIN  
3 Show_Salary(100,P_sal);  
4 dbms_output.put_line(P_sal);  
5 end;  
6 /  
Le salaire de MARTIN est de 6000  
6000
```

# Procédures

## Exemple 3: Afficher le salaire d'un employé

```
SQL> DECLARE P_Sal NUMBER;  
2 BEGIN  
3 Show_Salary(700,P_sal);  
4 END;  
5 /  
DECLARE P_Sal NUMBER;  
*  
ERROR at line 1:  
ORA-01403: no data found  
ORA-06512: at "SYSTEM.SHOW_SALARY", line 5  
ORA-06512: at line 3
```

# Procédures – gestion des erreurs

## Exemple 3: Afficher le salaire d'un employé

```
CREATE OR REPLACE PROCEDURE Show_Salary(P_Emp_Id Employee.Emp_Id%TYPE, P_Salary
OUT NUMBER ) IS
V_Emp_Name Employee.Emp_Nom%TYPE;
V_Salary Employee.Salary%TYPE;
BEGIN
    SELECT Salary, Emp_Nom INTO P_salary, V_Emp_Name
    FROM Employee WHERE Emp_Id=P_Emp_Id;
    DBMS_OUTPUT.PUT_LINE('Le salaire de ' || V_Emp_Name || ' est de ' || P_salary);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('L'employé ' || V_Emp_Name || ' n'existe pas');
END Show_Salary;
```

```
SQL> DECLARE P_sal NUMBER;
2 BEGIN
3 Show_Salary(700, P_sal);
4 END;
5 /
L'employe n'existe pas

PL/SQL procedure successfully completed.
```

# FONCTIONS

# Fonctions

Les fonctions retournent une valeur (number, integer, varchar2, booléen, date, ...).  
Son appel peut se faire via un ordre SQL Select, une procédure ou une fonction.

## Syntaxe :

```
CREATE OR REPLACE FUNCTION function_name(P_p1 type ...)  
RETURN type  
IS  
BEGIN  
instruction  
RETURN  
EXCEPTION  
END function_name;
```

# Fonctions

## Exemple : Afficher le salaire des employés

```
CREATE OR REPLACE FUNCTION Function_Show_Salary(P_Emp_Id  
NUMBER)  
RETURN NUMBER  
IS  
V_Salary NUMBER;  
BEGIN  
SELECT Salary INTO V_Salary FROM Employe WHERE Emp_Id=P_Emp_Id;  
RETURN V_Salary;  
END Function_Show_Salary;
```

```
SQL> DECLARE V_sal NUMBER;  
2 BEGIN  
3 V_Sal:= Function_Show_Salary(100);  
4 dbms_output.put_line(V_sal);  
5 END;  
6 /  
6000  
  
PL/SQL procedure successfully completed.
```

# Fonctions

## Exemple : Afficher le salaire des employés

```
SQL> SELECT Emp_Id, Function_Show_Salary(Emp_Id) FROM Employee;
```

| EMP_ID | FUNCTION_SHOW_SALARY(EMP_ID) |
|--------|------------------------------|
| 100    | 6000                         |
| 200    | 3000                         |
| 300    | 3000                         |
| 400    | 2500                         |



# STRUCTURES DE CONTROLE

# INSTRUCTION CONDITIONNELLE

**Instruction conditionnelle** : permet de faire des tests conditionnels

## Exemple

*IF condition THEN*

*ELSIF condition THEN*

*ELSIF condition THEN*

*ELSE séquence THEN → une seule et optionnelle*

*END IF;*

} Optionnelle  
↓

# INSTRUCTION CONDITIONNELLE

## Exemple

```
DECLARE V_nbre NUMBER:=10;  
BEGIN IF V_nbre>=10  
    THEN DBMS_OUTPUT.PUT_LINE('Je suis une valeur supérieure  
à 10');  
ELSIF V_nbre<10 AND V_nbre>=0  
    THEN DBMS_OUTPUT.PUT_LINE('Je suis une valeur inférieure à  
10 et supérieure à 0');  
ELSE DBMS_OUTPUT.PUT_LINE('Je suis une valeur négative');  
END IF;  
END;
```

# STRUCTURE CASE

**Structure case :** Permet aussi de mettre en place les tests conditionnels

## Syntaxe :

*Var*

*CASE Var*

*WHEN condion1 sur Var THEN instruction1;*

*WHEN condition2 sur Var THEN instruction2;*

*...*

*WHEN ConditionN sur Var THEN instructionN;*

*ELSE instruction*

*END CASE*

# STRUCTURE CASE

## Exemple

```
DECLARE V_nbre NUMBER:=13;  
BEGIN  
CASE V_nbre  
WHEN 10 THEN DBMS_OUTPUT.PUT_LINE('je suis une valeur égale à  
10');  
WHEN 11 THEN DBMS_OUTPUT.PUT_LINE('je suis une valeur égale à  
11');  
ELSE DBMS_OUTPUT.PUT_LINE('Je suis une valeur différente de 10  
et 11');  
END CASE;  
END;
```

Je suis une valeur différente de 10 et 11

Statement processed.

0,00 seconds

# STRUCTURE CASE

## Exemple

```
DECLARE V_nbre NUMBER:=10;  
BEGIN  
CASE  
WHEN V_nbre>=10 THEN DBMS_OUTPUT.PUT_LINE('je suis une  
valeur supérieure à 10');  
WHEN V_nbre<10 AND V_nbre>=0 THEN  
DBMS_OUTPUT.PUT_LINE('je suis une valeur inférieure à 10 et  
supérieure à 0');  
ELSE DBMS_OUTPUT.PUT_LINE('Je suis une valeur négative');  
END CASE;  
END;
```

je suis une valeur supérieure à 10

Statement processed.

0,00 seconds

# BOUCLE LOOP

**La boucle LOOP** : permet de faire une boucle avec une condition de sortie définit avec EXIT WHEN

## Syntaxe :

*LOOP*

*instruction*

*EXIT WHEN condition*

*END LOOP;*

## Exemple

```
DECLARE V_nbre NUMBER:=0;
```

```
BEGIN
```

```
LOOP
```

```
V_nbre:=V_nbre+1; DBMS_OUTPUT.PUT_LINE(V_nbre);
```

```
EXIT WHEN V_nbre=2;
```

```
END LOOP;
```

```
END;
```

```
1  
2
```

```
Statement processed.
```

# BOUCLE FOR

**La boucle FOR** : permet de faire une boucle avec une condition de sortie définir au départ

## Syntaxe :

```
FOR counter IN [REVERSE] borne_inf...borne_supérieure  
LOOP  
    instruction  
END LOOP;
```

## Exemple

```
DECLARE V_nbre NUMBER:=0;  
BEGIN  
FOR V_nbre in 1..3  
LOOP  
DBMS_OUTPUT.PUT_LINE(V_nbre);  
END LOOP;  
END;
```

Incrémentation automatique

```
1  
2  
3
```

```
Statement processed.
```



# BOUCLE WHILE

**La boucle WHILE:** permet de faire une boucle avec une évaluation de la condition au début de chaque itération

## **Syntaxe :**

```
WHILE condition  
LOOP  
Instruction  
END LOOP;
```

## **Exemple**

```
DECLARE V_nbre NUMBER:=0;  
BEGIN  
WHILE V_nbre <=2  
LOOP  
DBMS_OUTPUT.PUT_LINE(V_nbre); V_nbre:=V_nbre+1;  
END LOOP;  
END;
```

curseurs

# CURSEURS

Un curseur est une zone mémoire (un vecteur) dans laquelle les informations de traitement sont sauvegardées.

Il existe des curseurs **implicites** et **explicites**.

# CURSEURS

**CURSEUR IMPLICITE** : déclaré automatiquement par Oracle lors de l'exécution des requêtes.

Lors d'un SELECT, un seul enregistrement doit être résultat.

```
SQL> DECLARE V_Emp_Name VARCHAR2(50);
      2 BEGIN
      3 SELECT Emp_Nom INTO V_Emp_Name from eMPLOYE WHERE Dept_Id=20;
      4 DBMS_OUTPUT.PUT_LINE(V_Emp_Name);
      5 END;
      6 /
DECLARE V_Emp_Name VARCHAR2(50);
*
ERROR at line 1:
ORA-01422: exact fetch returns more than requested number of rows
ORA-06512: at line 3
```

# CURSEURS

**CURSEUR EXPLICITE** : déclaré et géré par les utilisateurs. Il permet de consulter plusieurs lignes et d'y effectuer des traitement sur chaque ligne.

**Syntaxe déclaration d'un curseur :**

**DECLARE CURSOR** *nom\_curseur [(P\_param1, P\_param2..)] IS  
SELECT statement;*

**Ouverture d'un curseur :**

*OPEN nom\_curseur[(P\_param1, P\_param2...)]*

**Accès aux lignes d'un curseur :**

*FETCH nom\_curseur INTO variable1, variable2...;*

**Fermeture du curseur :**

*CLOSE nom\_curseur*

# CURSEURS

La commande ***FETCH*** permet d'assigner le contenu de la ligne courante dans des variables et déplace le pointeur à la ligne suivante.

La commande **CLOSE** nom\_curseur permet de libérer l'espace mémoire alloué au curseur

## Attributs d'un curseur :

Nom\_curseur%**ROWCOUNT** : nombre de lignes affectées

Nom\_curseur%**FOUND** : prend la valeur TRUE si une ligne est trouvée, FALSE si non.

Nom\_curseur%**NOTFOUND** : prend la valeur TRUE si aucune ligne n'est retournée, FALSE si non

Nom\_curseur%**ISOPEN** =TRUE si le curseur est ouvert;

# CURSEURS

## Exemple :

```
DECLARE CURSOR C_Employe IS SELECT Emp_Nom, Salary FROM Employe WHERE
Dept_Id=20;
V_Emp_Nom Employe.Emp_Nom%TYPE;
V_Salary Employe.Salary%TYPE;
BEGIN
OPEN C_employe;
LOOP
FETCH C_employe INTO V_Emp_Nom, V_Salary;
DBMS_OUTPUT.PUT_LINE(V_Emp_Nom || ' a un salaire de '||V_Salary);
EXIT WHEN C_employe%NOTFOUND;
END LOOP;
CLOSE C_employe;
END
```

```
MARIE a un salaire de 3000
JEAN a un salaire de 2500

Statement processed.
```

# CURSEURS

PL/SQL permet de faire une boucle SQL spéciale pour les curseurs. Elle prend en charge les opérations du curseur : OPEN, FETCH, EXIT and CLOSE.

## Exemple :

```
DECLARE CURSOR C_Employe IS SELECT Emp_Nom, Salary FROM Employe
WHERE Dept_Id=20;
BEGIN
FOR V_rec IN C_Employe
LOOP → ouverture implicite du curseur
DBMS_OUTPUT.PUT_LINE(V_rec.Emp_Nom || ' a pour salaire ' || V_rec.Salary);
END LOOP; → fermeture implicite du curseur
END;
```

```
MARIE a un salaire de 3000
JEAN a un salaire de 2500

Statement processed.
```



# BULK BINDING

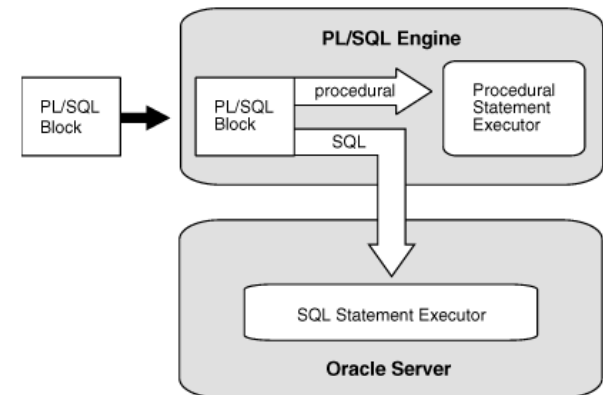
# BULK BINDING

Considérons le bloc ci-dessous, le traitement est fait ligne par ligne. Si il y'a 100 lignes à mettre à jour, il y'aura 100 échanges entre les moteurs SQL et PL/SQL

```
DECLARE V_iter NUMBER :=0;
BEGIN
FOR V_Emp IN(SELECT Emp_Id, Emp_Nom, Salary FROM Employe WHERE Dept_ID=20)
LOOP
V_iter :=V_iter+1;
UPDATE Employe SET Salary=Salary+100 Where Emp_Id=V_Emp.Emp_Id ;
DBMS_OUTPUT.PUT_LINE('Itération No ' || V_iter ||': Augmentation du salaire de '||V_Emp.Emp_Nom );
END LOOP;
END;
```

Itération No 1: Augmentation du salaire de MARIE  
Itération No 2: Augmentation du salaire de JEAN

Statement processed.



**Solution : BULK binds (liaison en masse)**

# BULK BINDING

Pour faire de la liaison en masse (BULK BINDING), deux fonctionnalités disponibles :

- **BULK COLLECT** permet de récupérer toutes les données en une seule extraction (du moteur SQL à PL/SQL) ;

**Syntaxe :** ... BULK COLLECT INTO collection\_name

Dans le cas d'un curseur :

FETCH cursor\_name BULK COLLECT INTO ... [LIMIT rows]

- **FOREALL** permet d'effectuer les opérations DML (INSERT, UPDATE, DELETE, MERGE) sur toutes les données d'une collection en une seule fois (de PL/SQL vers SQL)

**Syntaxe :** FOREALL index IN lower\_bound..upper\_bound  
sql\_statement (insert | update | delete );

# BULK BINDING

## Exemple :

```
DECLARE TYPE Type_Emp_Id IS TABLE OF Number;
TYPE Type_Emp_Nom IS TABLE OF Employee.Emp_Nom%TYPE;
V_Emp_Id Type_Emp_Id;
V_Emp_Nom Type_Emp_Nom;
V_index NUMBER;
BEGIN
SELECT Emp_Nom, Emp_Id BULK COLLECT INTO V_Emp_Nom,V_Emp_Id FROM Employee
WHERE Dept_ID=20;
DBMS_OUTPUT.PUT_LINE('Nombre de lignes retournées par le moteur SQL : ' ||
V_Emp_Id.COUNT);
FORALL V_index IN V_Emp_Nom.First..V_Emp_Nom.LAST
UPDATE Employee SET SALARY=SALARY+100 WHERE Emp_Id=V_Emp_Id(V_index);
DBMS_OUTPUT.PUT_LINE('Nombre de lignes mise à jour : ' || SQL%ROWCOUNT);
END;
```

---

```
Nombre de lignes retournées par le moteur SQL :2
Nombre de lignes mise à jour :2
```

```
Statement processed.
```

# TRIGGER

# TRIGGER

Un trigger encore appelé déclencheur est un traitement qui se déclenche suite à un évènement. Il permet une programmation évènementielle.

Deux types de trigger :

- **Applicatif** : créé et géré au niveau de l'application
- **Base de données** : stocké dans la base de données et associé aux évènements qui surviennent sur des tables.

Les évènements peuvent être :

- **DML** : insert, update, delete
- **DDL** : create, drop, alter ...
- **BASE** : erreur, logon...

# TRIGGER

Cas d'utilisations des triggers :

- Pour automatiser les traitements sur des évènements : déclencher une commande lorsque le stock atteint un certain seuil
- Garantir une propreté dans la base de données : garantir par exemple que le salaire d'un employé soit toujours compris dans une certaine fourchette.

Niveaux de déclenchement d'un trigger

- Niveau ligne (row level) : se déclenche pour chaque ligne
- Niveau instruction : se déclenche une seule fois

# TRIGGER

## Syntaxe :

```
CREATE OR REPLACE TRIGGER nom_trigger  
{BEFORE | AFTER} évènement  
ON nom_table  
REFERENCING OLD variable/NEW variable  
FOR {each row}  
DECLARE  
-- déclaration variable , curseur...  
BEGIN  
--traitement  
EXCEPTION  
--gestion des erreurs  
END nom_trigger;
```



# TRIGGER

**BEFORE | AFTER** : indique que le déclencheur doit être lancé avant | après l'exécution de l'évènement.

BEFORE si le trigger doit :

- déterminer si l'instruction DML est autorisée
- "fabriquer" la valeur d'une colonne pour pouvoir ensuite la mettre dans la.

AFTER si on a besoin que l'instruction DML soit terminée pour exécuter le corps du trigger

**INSERT , DELETE** : indique au déclencheur de s'exécuter lors d'une insertion ou d'une suppression dans la table

**UPDATE [of *colonne*]** : indique que le déclencheur doit être lancé lors de chaque mise à jour d'une des colonnes spécifiées. Si [of *colonne*] n'est pas précisé, n'importe quelle colonne de la table modifiée provoque le déclenchement du trigger

# TRIGGER

**ON nom\_table** : désigne le nom de la table associé à son schéma pour lequel le trigger a été créé.

**FOR EACH ROW** : si spécifié => trigger de ligne. Le trigger se déclenche pour toutes les lignes de la mise à jour. Si non spécifié => trigger d'instruction qui se déclenche une seule fois.

**REFERENCING OLD variable|NEW variable** : permet de renommer les variables de corrélation OLD et NEW

**WHEN (condition)** : spécifie une restriction sur le trigger. La restriction est une condition SQL qui doit être satisfaite pour que le trigger se déclenche.

# TRIGGER – variables de corrélation

Pour un trigger niveau ligne, on peut avoir besoin d'accéder aux données du tuple en cours de manipulation. Cela se fait via deux records **:old** et **:new** qui ont la même structure que la table sur laquelle le trigger est défini.

| Evènement | :OLD               | :NEW                        |
|-----------|--------------------|-----------------------------|
| INSERT    | NULL               | Valeur en cours d'insertion |
| UPDATE    | Ancienne valeur    | Nouvelle valeur             |
| DELETE    | Valeur a supprimée | NULL                        |

! Ne sont pas des variables globales

# TRIGGER

## **Before trigger :**

On peut écrire dans :new valeur mais on ne peut pas modifier :old valeur.

## **After trigger :**

On ne peut écrire ni dans :new valeur ni dans :old valeur

Erreur obtenue en cas de mauvaise modification :

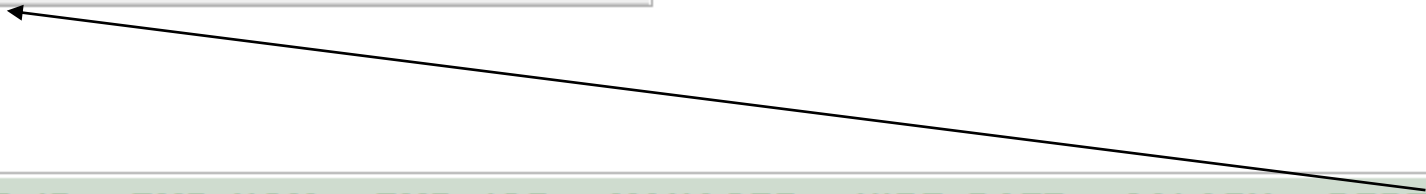
ORA-04084: cannot change NEW values for this trigger type

# EXEMPLES

Considérons les tables ci-dessous : Table Employee et Table Department

| DEPT_ID | DEPT_NAME  | COUNTRY |
|---------|------------|---------|
| 10      | ACCOUNTING | PARIS   |
| 20      | RESEARCH   | PARIS   |
| 30      | SALES      | LYON    |
| 40      | OPERATIONS | PARIS   |

| EMP_ID | EMP_NOM | EMP_JOB   | MANAGER | HIRE_DATE | SALARY | DEPT_ID |
|--------|---------|-----------|---------|-----------|--------|---------|
| 100    | MARTIN  | PRESIDENT | -       | 22/06/01  | 6000   | 10      |
| 200    | DUPONT  | MANAGER   | 100     | 22/10/01  | 3000   | 30      |
| 300    | MARIE   | MANAGER   | 100     | 22/11/01  | 3000   | 20      |
| 400    | JEAN    | ANALYST   | 300     | 30/11/01  | 2500   | 20      |



# TRIGGER

Exemple 1 :

```
CREATE OR REPLACE TRIGGER Verifie_salaire  
BEFORE INSERT  
ON Employe  
FOR EACH ROW  
WHEN (new.salary < 1300)  
BEGIN  
raise_application_error(-20000, 'salaire incorrect, le salaire doit etre  
supérieur au SMIC');  
END Verifie_salaire;
```

# TRIGGER

Exemple 1 :

## Insertion d'une nouvelle ligne

```
INSERT INTO Employe(Emp_Id , Emp_Nom , Emp_Job , Manager ,  
Hire_Date , Salary , Dept_Id) VALUES (600, 'ANDREA', 'ANALYST'," ,  
to_date('2010-07-22','yyyy-MM-dd'), 1000,10) ;
```

```
ORA-20000: salaire incorrect, le salaire doit etre supérieur au SMIC  
ORA-06512: à "SYSTEM.VERIFIE_SALAIRE", ligne 2  
ORA-04088: erreur lors d'exécution du déclencheur 'SYSTEM.VERIFIE_SALAIRE'
```

La ligne n'est pas ajoutée à la table

BEFORE INSERT : détermine si l'instruction DML est autorisée

# TRIGGER

## EXEMPLE 2 : historisation des données lors de la suppression

```
CREATE OR REPLACE TRIGGER Archive_Employe  
AFTER DELETE  
ON Employe  
FOR EACH ROW  
BEGIN  
INSERT INTO Archive_Employe VALUES (:old.Emp_id, :old.Emp_Nom,  
:old.Emp_Job,:old.Manager, :old.Hire_Date,:old.Salary, :old.Dept_Id);  
END Archive_Employe;
```

Suppression d'un employé : *DELETE FROM Employe where emp\_id=500;*

Contenu de la table Archive\_employe

| EMP_ID | EMP_NAME | EMP_JOB | MANAGER | HIRE_DATE | SALARY | DEPT_ID |
|--------|----------|---------|---------|-----------|--------|---------|
| 500    | LOUIS    | ANALYST | -       | 22/06/10  | 2600   | 10      |



# TRIGGER

Il est possible pour un déclencheur de tester l'évènement déclencheur avec les prédicats:

- If inserting then ...
- If deleting then ...
- If updating then ...
- If updating [(colonne)] then ...

# TRIGGER

## Exemple

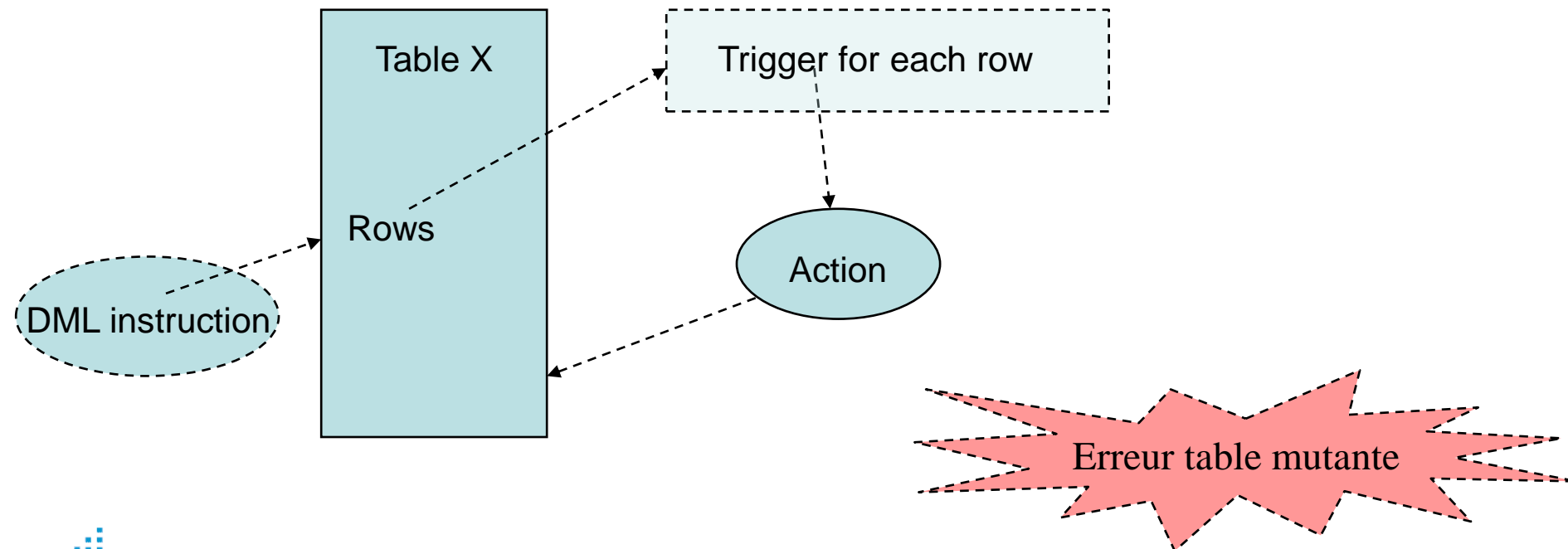
```
CREATE OR REPLACE TRIGGER test_even
AFTER INSERT OR UPDATE of SALARY OR DELETE
ON EMPLOYE
FOR EACH ROW
BEGIN
IF INSERTING THEN
    IF :new.SALARY<1300 THEN
        DBMS_OUTPUT.PUT_LINE( 'attention le salaire est inferieur a 1300');
    END IF;
END IF;
IF UPDATING THEN
    IF :new.SALARY<1300 THEN
        DBMS_OUTPUT.PUT_LINE( 'attention le salaire est inferieur a 1300');
    END IF;
END IF;
IF DELETING THEN DBMS_OUTPUT.PUT_LINE('suppression');
END IF;
end test_even;
```

# TRIGGER – Table mutante

Une **table mutante** est une table en cours de modification du fait d'une instruction DML (update, delete, insert).

Considérons le schema ci-dessous.

La table X est appelée table **mutante**.



# TRIGGER – Table mutante

! Un trigger ne peut pas modifier la table concernée par l'instruction qui a déclenché le trigger.

Cet contrainte évite que l'on ai des lectures incohérentes

## Exemple :

```
CREATE OR REPLACE TRIGGER Test_Table_Mutante
BEFORE UPDATE
ON EMPLOYE
FOR EACH ROW
BEGIN
UPDATE EMPLOYE SET SALARY = 3000 WHERE Emp_Id=100;
END Test_Table_Mutante;
```

```
UPDATE EMPLOYE SET SALARY = 3000 WHERE Emp_Id=100
*
ERROR at line 1:
ORA-04091: table SYSTEM.EMPLOYE is mutating, trigger/function may not
see it
```

# TRIGGER – Oracle

On peut créer des triggers pour des événements au niveau de la base de données Oracle

L'option After pour des événements comme **STARTUP**,  
**SERVERERROR**, **LOGON**

L'option Before pour des événements comme **LOGOFF**,  
**SHUTDOWN**

**Exemple :**

```
CREATE OR REPLACE TRIGGER Trace AFTER LOGON
BEGIN
INSERT INTO Trace_Table VALUES(user_name, sysdate);
END;
```

# TRIGGER

Gestion des triggers :

- **DROP TRIGGER trigger\_name** : pour supprimer un trigger
- **ALTER TRGGIER trigger\_name {ENABLE / DISABLE}**  
pour activer ou desactiver un trigger. Lorsqu'un trigger est créé, il est automatiquement activer. Desactiver un trigger ne le supprime pas de la base.
- **ALTER TABLE table\_name {ENABLE / DISABLE} ALL TRIGGERS** : permet d'activer ou desactiver tous les triggers d'une table

# PACKAGES

# PACKAGE

Un package est un schema qui regroupe des objects PL/SQL(type, procédures, fonctions...) logiquement liés.

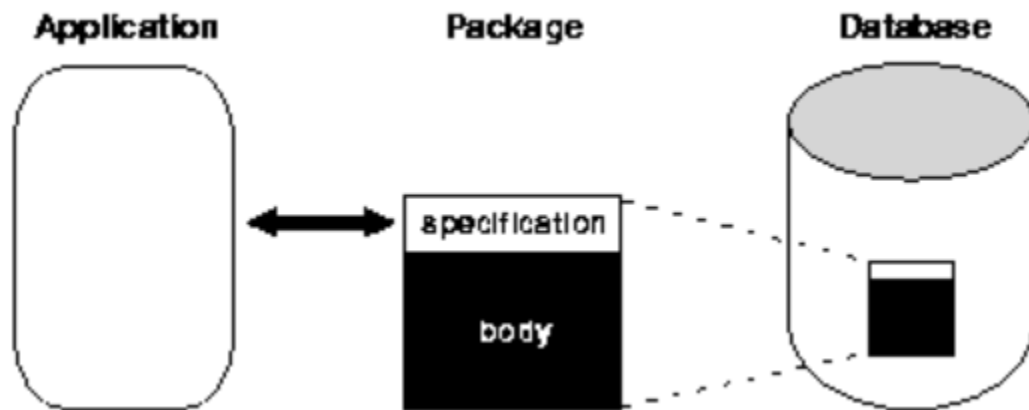
Un package a deux parties :

- **Specification** : permet de déclarer les types, variables, procédures, curseurs...
- **Body** : permet d'implémenter les éléments déclarés dans la specification notamment les curseurs, procédures, fonctions.



# PACKAGE

La partie spécification est l'interface avec l'application



# PACKAGE

## Spécification :

```
CREATE OR REPLACE PACKAGE package_name
```

```
AS
```

```
--Declaration variables, curseurs,exceptions...
```

```
--Prototypes des procédures , fonctions
```

```
END package_name
```

# PACKAGE

## **BODY :**

CREATE OR REPLACE PACKAGE **BODY** package\_name

AS

*Specifiction des fonctions, procédures*

BEGIN

*Commandes a exécuter*

END package\_name

# PACKAGE

## EXEMPLE :

```
CREATE OR REPLACE PACKAGE pkg_Gestion_Employe  
AS  
-- declaration des variables globales  
v_nbre_employe NUMBER;  
V_nbre_employe_department NUMBER;  
--declaration des prototypes  
FUNCTION Affiche_nbre_employe RETURN NUMBER;  
FUNCTION Affiche_nbre_emp_dep(Dept_id NUMBER) RETURN NUMBER;  
END pkg_Gestion_Employe;
```

# PACKAGE

## EXEMPLE :

```
CREATE OR REPLACE PACKAGE BODY pkg_Gestion_Employe  
IS
```

```
FUNCTION Affiche_nbre_employe  
RETURN NUMBER  
IS
```

```
v_total NUMBER;  
BEGIN  
SELECT COUNT(*) INTO v_total FROM Employe;  
RETURN v_total;  
END Affiche_nbre_employe;
```

# PACKAGE

```
FUNCTION Affiche_nbre_emp_dep(Dept_id NUMBER)  
RETURN NUMBER  
IS  
v_total NUMBER;  
BEGIN  
SELECT COUNT(*) INTO v_total FROM Employe where Dept_Id=Dept_id;  
RETURN v_total;  
END Affiche_nbre_emp_dep;  
BEGIN  
NULL ;  
END pkg_Gestion_Employe;
```

# PACKAGE

## EXEMPLE :

```
DECLARE var NUMBER :=0;  
BEGIN  
var:=pkg_Gestion_Employe.Affiche_nbre_emp_dep(20);  
DBMS_OUTPUT.PUT_LINE(var);  
END
```

# SQL DYNAMIQUE



# SQL Dynamique

Rôle du SQL Dynamique :

- Exécuter des ordres DDL (create, drop, alter...) dans un bloc PL/SQL
- Jusqu'à présent, tous les ordres SQL écrits dans du code PL/SQL étaient statiques. Il y a des cas où l'ordre SQL n'est connu qu'à l'exécution. Par exemple lorsque l'on ne connaît pas la table du select. Un ordre SQL peut être stocké dans une chaîne de caractères puis exécuté.



Ordre dynamique analysé à chaque exécution

# SQL Dynamique : EXECUTE IMMEDIATE

**EXECUTE IMMEDIATE** permet d'analyser et d'exécuter immédiatement une instruction SQL Dynamique ou un bloc anonyme

Syntaxe :

```
EXECUTE IMMEDIATE dynamic_string  
[INTO {define_variable, define_variable ... | record}]  
[USING [IN | OUT | IN OUT] bind_argument]  
[{RETURNING | RETURN } INTO bind_argument]
```

# SQL Dynamique : EXECUTE IMMEDIATE

Exemple :

*DECLARE*

*sql\_stmt* VARCHAR2(200);

*v\_id* NUMBER :=2;

*v\_Cours* VARCHAR2(50):= 'Mathématiques';

*V\_id\_ajoute* NUMBER;

*V\_cours\_ajoute* VARCHAR2(50);

*BEGIN*

*EXECUTE IMMEDIATE* 'CREATE TABLE Cours (id NUMBER, Cours VARCHAR2(50))';

*sql\_stmt* := 'INSERT INTO Cours VALUES (1, "Bases de données)";

*EXECUTE IMMEDIATE* *sql\_stmt*;

*sql\_stmt* := 'INSERT INTO Cours VALUES (:1, :2) RETURNING id , cours INTO :3, :4';

*EXECUTE IMMEDIATE* *sql\_stmt* USING *v\_id*, *v\_cours* RETURNING INTO  
*v\_id\_ajoute*,*v\_cours\_ajoute*;

*dbms\_output.put\_line*('Le cours ' || *v\_cours\_ajoute* || ' a été ajouté avec l'id ' ||*v\_id\_ajoute*);

*EXECUTE IMMEDIATE* 'DROP TABLE Cours';

*END*;

Le cours Mathématiques a été ajouté avec l'id 2

Statement processed.

# DESCRIBE

- **DESCRIBE** : cette fonction permet de donner la description des objets.

```
SQL> DESCRIBE Show_Salary;
```

```
PROCEDURE Show_Salary
```

| Argument Name | Type       | In/Out | Default? |
|---------------|------------|--------|----------|
| P_EMP_ID      | NUMBER(10) | IN     |          |
| P_SALARY      | NUMBER     | OUT    |          |

# WRAP

- **WRAP** : Permet de coder le code source PL/SQL; il permet par exemple de partager un fichier sans toute fois donner accès au code.

# Quelques vue ORACLE

- **USER\_OBJECTS** : contient les objets oracle de l'utilisateur
- **USER\_SOURCE** : contient le code source des sous programmes de l'utilisateur. Cette vue permet de retrouver le code des programmes stockés
- **USER\_PROCEDURE** : contient toutes les fonctions et procédures de l'utilisateur.
- **USER\_CONSTRAINTS** : contient toutes les contraintes définies sur les tables de l'utilisateur en cours.

Vous trouverez la liste des vues systèmes oracle sur le site [docs.oracle.com](https://docs.oracle.com)

# Fin